CRAM Format Specification

Author: Josh Green <josh@resonance.org>
Revision: 1.2
Date: May 19, 2007
URL: http://cram.resonance.org

Table of Contents

1 Revision history

Revision 1.2 – May 19, 2007

- Switched to using non variable length encoded integers for fields, breaking backwards compatibility.
- Format version reset to **v1** (since it wasn't stored properly in older versions anyways)
- Previous versions (1-3) renamed to (0x81 – 0x83), to indicate the falsely stored version values
- DocType field should now come directly after the EBML chunk, to aid in file identification
- Added CRAMInfo chunk to contain Flags, Software and Comment fields. This was done so that the DocType signature is always a fixed offset (EBML chunk size always 1 byte), to aid in file identification.
- Added section on file identification including Unix **file** utility magic.
- Removed AudioMD5 chunk and added FileMD5 chunk. This means that FileMD5 now requires a separate pass (needed anyways for out of order binary chunk storage). Section added in FileChunk description on MD5 sums.
- WavPackSplitChunk and FlacSplitChunk no longer used at the FileData level, since WavPackChunk and FlacChunk can be used for both split or non-split audio.
- CramInfo and Software field are now required, set max size of Software field to 64 bytes and Comment to 4096 bytes.
- EBMLVersion and EBMLReadVersion chunks no longer used (DocTypeVersion covers it).
- Added FileInfo chunk to contain information related fields.
- Added optional CRC-32 for CramInfo and FileInfo chunks.
- Chunk order should now be considered fixed to make it easier on the decoder implementation

Revision 1.1 – May 11, 2007

- Updated EBML ID tables to not use indentation (didn't work in HTML export)
- Added SetAudioEntry and NextAudioEntry chunks for chaining audio segments and section on grouping audio segments
- Clarification of RawSize for SampleSplit audio entries
- Added VerbatimChunk type
- Added signed int data type
- Other minor clarifications

Revision 1.0 – May 5, 2007

- Initial release of document describing **CRAM** format version 4 (actually v1).

2 About

CRAM is an acronym for "**C**ompress hyb**R**id **A**udio **M**edia". It is an open file format that was created to handle the compression of files containing audio and binary. Better compression of audio can often be achieved by codecs specifically designed for audio (such as **FLAC** or **WavPack**) than a binary compressor. CRAM utilizes audio codecs for audio (FLAC and WavPack currently) and binary compressors for binary (**bzlib** and **zlib** currently) which often yields a much higher compression ratio than if a purely binary compressor was used. Format specific encoders are created to handle the compression of a specific data format, but the decoder is generic, which means future support for the compression of other formats is backwards compatible. CRAM is currently specifically used for instrument file compression (such as **SoundFont©**[1], **DLS** and **GigaSampler©**[2]), but is not limited to this purpose.

3 Features

- Open standard with **LGPL** licensed reference software implementation (**libInstPatch**)
- Based on **EBML** (a binary XML like format) which provides flexibility in extending the format
- Compression of multiple files with paths in a single archive with time preservation
- Extensive audio format support including: 8/16/24/32 bit integer sample widths, floating point audio (**WavPack** only), stereo or mono (support for up to 8 audio channels is planned), signed or unsigned, big endian or little endian
- Support for split stereo (mono audio segment pairs stored separately) and split sample data (such as 24 bit support in **SoundFont©** files where 16 bit and 8 bit sample width portions are stored separately).
- Lossy/lossless hybrid mode (**WavPack** only) where a much smaller lossy CRAM file can be stored with a separate correction file which when combined results in the original lossless data. The combined sizes of the files are only slightly larger than the equivalent lossless CRAM file. This allows for the creation of a smaller preview of a potentially large file.

4 File extensions

There are 3 file extensions used with CRAM.

1. **.cram** – Used for lossless CRAM files
2. **.craml** – Lossy CRAM file
3. **.cramc** – CRAM correction file

5 File identification

A CRAM file can be identified by the four EBML bytes 0x1A 0x45 0xDF 0xA3 at the beginning of the file. At offset 5 will be found the DocType ID consisting of 2 bytes 0x42 0x82. At offset 8 will be found the characters "CRAM", "CRAML" or "CRAMC" for a lossless, lossy or correction CRAM file respectively. Below is the corresponding magic for the Unix **file** utility.

1 "SoundFont" is a registered trademark of E-mu Systems, Inc.
2 GigaSampler is a registered trademark of Nemesys Music. Technology, Inc.

```
# EBML id:
0               belong          0x1a45dfa3
# DocType id:
>5              beshort         0x4282
# DocType contents:
>>8             string          CRAM        CRAM archive data
>>8             string          CRAML       CRAM lossy archive data
>>8             string          CRAMC       CRAM correction data
```

6 EBML format

The following is an overview of the **EBML** format, which is likely enough information to utilize the CRAM format. More information can be found though, on the EBML website.

6.1 Variable length integers

One feature of EBML is variable length encoded integers. This is used for EBML chunk IDs and sizes. Of note is that **uint** and **int** field values are not variable length encoded (the length of the integer can be inferred from the chunk size, this was also broken in previous CRAM format versions). EBML stores integer values in **big endian** byte order. The count of 0 bits in the first byte (starting from the most significant bit) + 1 determines the number of bytes that the integer is composed of. The following table illustrates this:

Size	Range start (hex)	Range end (hex)	Effective range (decimal)
1	0x80	0xFE	0 to 2^7 - 2
2	0x4000	0x7FFE	0 to 2^14 - 2
3	0x200000	0x3FFFFE	0 to 2^21 - 2
4	0x10000000	0x1FFFFFFE	0 to 2^28 - 2
5	0x0800000000	0x0FFFFFFFFE	0 to 2^35 - 2
6	0x040000000000	0x07FFFFFFFFFE	0 to 2^42 - 2
7	0x02000000000000	0x03FFFFFFFFFFFE	0 to 2^49 - 2
8	0x0100000000000000	0x01FFFFFFFFFFFFFE	0 to 2^56 - 2

For example: A value of 64 decimal (0x40 hex) could be stored as a single byte 0xC0, as a 2 byte integer 0x4040, as a 3 byte integer 0x200040, etc. A larger value such as 8192 decimal (0x2000 hex) would require at least 2 bytes to encode (0x6000). In this way values up to **2^{56} - 2** can be stored in a space efficient manner (thats 72057594037927934 decimal).

6.2 EBML chunks

An EBML chunk can be illustrated by the following table:

Field Name	Data Type	Example
ID	Variable length integer	(1A)(45)(DF)(A3)
Size	Variable length integer	(C0)

Data	Arbitrary binary data of **Size** bytes	64 bytes follow

To summarize: An EBML chunk consists of an ID, Size and the actual data of the chunk. The ID and Size are variable length integers and the data in the chunk is arbitrary, although a format using EBML (such as CRAM) defines specific IDs for particular types of data.

Example description

In the above example the EBML ID is defined by 4 bytes (1A)(45)(DF)(A3), which is the EBML top level document ID. Following the ID is the size field which is stored as a single byte (C0) byte which has an effective value of 64 decimal. The payload of the chunk then follows at 64 bytes in size (in this case the payload would consist of additional EBML chunks, since the ID corresponds with the toplevel EBML document chunk).

6.3 Data types

There are a number of predefined data types for use in EBML files and there are some additional ones that CRAM uses. The data types used by CRAM include:

Data Type Name	**Description**
uint	Integer from 1 to 8 bytes (not variable length encoded, since size can already be inferred from chunk size).
int	Like uint but the value is interpreted as a signed integer.
string	Printable ASCII (0x20 to 0x7E)
UTF8	Unicode string
elements	Embedded EBML elements
binary	Block of binary data
MD5	16 byte (128 bit) MD5 sum
CRC32	4 byte integer CRC-32 signature

6.4 CRC32 checksums

CRC32 checksums can be optionally used for the CramInfo and FileInfo chunks. It is recommended that they are present in order to catch corruption of information, although decoders should attempt to continue even if the CRC fails. The compressed audio/binary data itself is protected by MD5 checksums. The file relocation tables use an Adler-32 checksum in the case of **zlib** (bzlib has its own CRC32 built in).

6.5 Errata

Versions **0x81** through **0x83** of the CRAM format were broken. All integer field values were stored as variable length encoded integers (what is used for the CRAM chunk IDs and sizes). This was not only unnecessary (integer length can be inferred from chunk size) but also contrary to EBML spec. Since CRAM was not yet widely (if at all) used, it was decided to break backwards compatibility and set things right. Integer fields are now just stored as regular big endian values.

Older CRAM files can be detected by the falsely written EBMLVersion chunk (should be the first chunk within the top EBML chunk). The value 0x81 (129 decimal) will be found, which corresponds to a variable length encoded value of 1. The correct value is just a single byte containing 0x01.

7 CRAM format reference

- First column defines if field is required ('*' indicates requirement).
- (XX) defines a hex byte value
- (**vN**) indicates a specific CRAM format version requirement.
- Refer to Data Types for details on the data types in the following tables.
- “[]” is used to indicate one or more values (an array)
- '|' is used to illustrate the embedded tree structure of the EBML chunks
- This color indicates chunks which contain embedded chunks
- This color indicates the chunk is further defined in another table

7.1 CRAM file format

	Field Name	EBML ID	Data Type	Description
*	EBML chunk	(1A)(45)(DF)(A3)	elements	EBML document chunk.
*	| DocType	(42)(82)	string	“CRAM”, “CRAML”, or “CRAMC
*	| DocTypeVersion	(42)(87)	uint	CRAM format version = 1.
*	| DocTypeReadVersion	(42)(85)	uint	CRAM read version required = 1.
*	CramInfo chunk	(41)(80)	elements	Information for entire CRAM file
*	| Software	(61)(3C)	string	Software name and version
	| Flags	(61)(31)	uint	See Flags
	| Comment	(61)(45)	UTF8	Comment with '\n' newline chars.
	| CRC-32	(BF)	CRC32	Checksum of CramInfo data
*	**FileChunks**[]	(41)(81)	elements	One or more file chunks
*	FileEnd	(41)(8C)	empty	End of file chunk.

7.2 FileChunk

	Field Name	EBML ID	Data Type	Description
*	FileChunk	(41)(81)	elements	Defines a file in a CRAM archive.
*	| FileInfo	(41)(84)	elements	File information chunk
*	| | FileSize	(61)(53)	uint	Original file size.
*	| | FileName	(61)(57)	UTF8	Relative path delimited by '/'s.
	| | FileDate	(61)(5C)	uint	Unix 32 bit timestamp.
	| | Flags	(61)(31)	uint	See Flags
	| | CRC-32	(BF)	CRC32	Checksum of FileInfo data
*	| FileData	(41)(95)	elements	Stores compressed audio and binary

	Field Name	EBML ID	Data Type	Description
	| | BinaryChunk	(C2)	binary	Compressed binary chunk
	| | FlacChunk	(D3)	binary	Compressed FLAC audio chunk
	| | WavPackChunk	(D9)	binary	Compressed WavPack audio chunk
	| | VerbatimChunk	(C7)	binary	Verbatim uncompressed data
	| | SetAudioEntry	(C4)	int	See Grouping audio segments
*	| **RelocTable**	(41)(A5)	elements	Relocation table (compressed)
	| RelocTableChkSum	(41)(A9)	uint	Adler32 RelocTable checksum (zlib only)
	| FileMD5	(41)(AA)	MD5	MD5 of original file
	| BinaryMD5	(41)(AC)	MD5	MD5 sum of original binary data

7.3 Relocation table

	Field Name	EBML ID	Data Type	Description
*	RelocTable	(41)(A5)	elements	Relocation table (compressed)
	| BinaryChunk	(C2)	elements	Binary data relocation entry
*	| | RawSize	(61)(63)	uint	Uncompressed size in bytes
	| FlacChunk	(D3)	elements	FLAC audio relocation entry
*	| | RawSize	(61)(63)	uint	Uncompressed size in bytes
	| | Flags	(61)(31)	uint	See Flags
	| | NextAudioEntry	(C5)	uint	See Grouping audio segments
	| FlacSplitChunk	(D7)	elements	FLAC split audio relocation entry
*	| | RawSize	(61)(63)	uint	Size of 1st split segment in bytes
	| | Flags	(61)(31)	uint	See Flags
	| | NextAudioEntry	(C5)	uint	See Grouping audio segments
	| | ChanCount	(61)(65)	uint	Channel count (default = 2)
	| | SampleSplit	(61)(69)	uint	Sample width split
	| | Offsets	(61)(66)	uint[]	Offset of each audio chunk.
	| WavPackChunk	(D9)	elements	WavPack audio relocation entry
*	| | RawSize	(61)(63)	uint	Uncompressed size in bytes
	| | Flags	(61)(31)	uint	See Flags
	| | NextAudioEntry	(C5)	uint	See Grouping audio segments
	| | SmoothLoop	(61)(73)	uint[2]	LoopStart, LoopSize
	| WavPackSplitChunk	(DD)	elements	WavPack split audio relocation entry
*	| | RawSize	(61)(63)	uint	Size of 1st split segment in bytes

	Field Name	EBML ID	Data Type	Description
	| | Flags	(61)(31)	uint	See Flags
	| | NextAudioEntry	(C5)	uint	See Grouping audio segments
	| | ChanCount	(61)(65)	uint	Channel count (default = 2, stereo)
	| | SampleSplit	(61)(69)	uint	Sample width split
	| | Offsets	(61)(66)	uint[]	Offset of each audio chunk
	| | SmoothLoop	(61)(73)	uint[2]	LoopStart, LoopSize

7.4 Chunks listed by EBML ID

Re-listing of all chunks, sorted by EBML ID.

	Field Name	EBML ID	Data Type	Description
*	EBML chunk	(1A)(45)(DF)(A3)	elements	EBML document chunk.
*	CramInfo chunk	(41)(80)	elements	Information for entire CRAM file
*	FileChunk	(41)(81)	elements	Defines a file in a CRAM archive.
*	FileInfo	(41)(84)	elements	File information chunk
*	FileEnd	(41)(8C)	empty	End of file chunk.
*	FileData	(41)(95)	elements	Stores compressed audio and binary
*	RelocTable	(41)(A5)	elements	Relocation table (compressed)
	RelocTableChkSum	(41)(A9)	uint	RelocTable checksum (zlib only)
	FileMD5	(41)(AA)	MD5	MD5 of original file
	BinaryMD5	(41)(AC)	MD5	MD5 sum of original binary data
*	DocType	(42)(82)	string	“CRAM”, “CRAML”, or “CRAMC”
*	DocTypeReadVersion	(42)(85)	uint	CRAM read version required
*	DocTypeVersion	(42)(87)	uint	CRAM format version
	Flags	(61)(31)	uint	See Flags
*	Software	(61)(3C)	string	Software name and version
	Comment	(61)(45)	UTF8	Comment with '\n' newline chars.
*	FileSize	(61)(53)	uint	Original file size.
*	FileName	(61)(57)	UTF8	Relative path delimited by '/'s.
	FileDate	(61)(5C)	uint	Unix 32 bit timestamp.
*	RawSize	(61)(63)	uint	Uncompressed size in bytes
	ChanCount	(61)(65)	uint	Channel count (default = 2, stereo)
	Offsets	(61)(66)	uint[]	Offset of each split segment
	SampleSplit	(61)(69)	uint	Sample width split

	Field Name	EBML ID	Data Type	Description
	SmoothLoop	(61)(73)	uint[2]	LoopStart, LoopSize
	CRC-32	(BF)	CRC32	A CRC-32 checksum of chunk data
	BinaryChunk	(C2)	bin/elements	Binary data or relocation entry
	SetAudioEntry	(C4)	uint	Offset current audio relocation entry
	NextAudioEntry	(C5)	int	Offset to next relocation entry
	VerbatimChunk	(C7)	bin/elements	Verbatim data or relocation entry
	FlacChunk	(D3)	bin/elements	FLAC data or relocation entry
	FlacSplitChunk	(D7)	bin/elements	FLAC split relocation entry
	WavPackChunk	(D9)	bin/elements	WavPack data or relocation entry
	WavPackSplitChunk	(DD)	bin/elements	WavPack split relocation entry

8 Cram format description

A CRAM file consists of an EBML chunk, optional EBMLInfo chunk, one or more FileChunks and a FileEnd terminator chunk.

8.1 CRAM format version history

The CRAM format version is updated whenever any changes occur to the defined format. If any new features are used in a CRAM file that would cause the parser to break or make it impossible for previous decoder versions to reconstruct the compressed files the appropriate format version required should be written. Note that the lowest compatible read version should be used in all cases, so only the use of new features should warrant an increment in the stored read version.

NOTE: Versions **v0x81** through **v0x83** are considered pre-formats. They were broken in regards to storing integer fields (including the DocTypeVersion itself, which is indicated by the 0x81, 0x82 and 0x83). For this reason, and the fact that backwards compatibility was broken, the version was restarted at version 1. This shouldn't be much of an issue though, since CRAM was not yet in wide use.

- **v1** – 2007-04-29
  - Backwards compatibility completely broken
  - Integer fields are now stored non UTF encoded (wasn't necessary to begin with)
  - Added AudioEntryOfs and NextEntryOfs to allow multiple audio segments to be encoded together
  - Added WavPack support, including hybrid lossy/lossless mode
  - Added bzip2 support
  - Added Verbatim data chunk
  - Split chunk types no longer used in FileData chunk
  - Removed AudioMD5 and added FileMD5 (MD5 on entire original file). MD5 now requires a separate pass (needed anyways due to sample grouping).
- **v0x83** – 2006-07-31

    - sampleSplit chunk type added to support 24 bit SoundFont files
- **v0x82** - 2005-03-20 - completely backwards compatible with v1
    - Added Software and Comment fields to CRAM header
    - Future defined chunks should also be expected at the fileChunk level (backwards compatibility with format v1 will be broken if a new chunk is added at this level)
- **v0x81** - First version

8.2 EBML chunk

The **EBML chunk** defines the file as an EBML file with one of 3 document types: "CRAM", "CRAML" or "CRAMC" for a lossless, lossy and correction CRAM file types respectively. In addition this chunk defines the CRAM format version of the encoder used to write the file and the required format version to read the file. All fields in the EBML chunk should be in the same order indicated and the EBML chunk size value should always be 1 byte in length. This is to aid in identifying CRAM files. The EBMLVersion and EBMLReadVersion chunks which are part of the EBML standard both default to a value of 1. This is unlikely to change for the CRAM format, so they have been omitted from the standard. If for some reason a newer version of EBML is used (highly unlikely), it would be part of a new CRAM version (covered by DocTypeReadVersion).

8.3 CramInfo chunk

Required chunk containing data which applies to the entire CRAM archive. Currently includes 3 fields: Software, Flags, and Comment. The Software field is required and is a UTF-8 string which describes the software used to write the CRAM file (example: "libInstPatch 1.0.0"),. it should not exceed 64 bytes. The comment field is an optional UTF-8 encoded string with '\n' line terminators. Size of Comment string should be limited to a max of 4096 bytes. When displaying this field, a newline should be added if necessary to the end.

Example comment

Downloaded from Resonance Instrument Database\n http://sounds.resonance.org

The Flags field is optional and defines the global flags value for the CRAM archive. An optional CRC-32 field may be present and defines the CRC-32 checksum value for all other data (besides the CRC chunk and value itself) in the CramInfo chunk. CRAM parsers should issue a warning if this CRC fails, but should attempt to continue.

8.4 Flags

The flags field is used throughout the CRAM format. Not all flags are valid at every level of the format though. Note that the binary compressor type is actually a 2 bit field defining up to 4 different binary compressor types (only 2 used currently). The given flag may be used at all levels where "Flags" is defined unless otherwise noted. A Flags chunk at a more embedded level overrides that of a higher level.

Flag Name	**Value**	**Description**
Binary compressor	0x03 mask (2 bit enum)	Binary compressor used 0 = **zlib**, 1 = **bzlib** (**v4**), 2/3 = **reserved** Only valid in EBML and fileChunk levels, this field is ignored at other levels.
Big endian audio	0x04	If set to 1, indicates that audio is big endian.
Unsigned audio	0x08	If set to 1, indicates that audio is unsigned.

8.5 FileChunk

The **FileChunk** defines a single file in a CRAM archive. It contains, in order, the FileInfo, FileData, RelocTable, RelocTableChkSum, FileMD5 and BinaryMD5 chunks.

The FileInfo chunk contains, in order, the required FileSize and FileName chunks and optional FileDate, Flags and CRC-32 checksum. The FileSize field stores the original file size. FileName contains a UTF-8 encoded file name which can include a relative '/' delimited path for storing paths in the archive. Parsers should issue a warning if the CRC-32 of the FileInfo fails, but continue anyways.

The FileData chunk contains zero or more compressed binary, uncompressed binary and audio chunks. Types of binary chunks include: BinaryChunk and VerbatimChunk. The former is for compressing blocks of binary data (with bzlib or zlib) and the latter is for just storing raw data as is (used for example, if it is already compressed).

The types of audio chunks include: FlacChunk and WavPackChunk.

The FileMD5 chunk stores the 16 byte MD5 sum of the original file (binary and audio). The BinaryMD5 is the MD5 sum of the original uncompressed binary data concatenated together. This allows for data integrity checks of only the binary data in the event that the audio data will be streamed (removes the need to reconstruct the whole file) or in the case of a CRAML file where the audio data is lossy.

8.6 Grouping audio segments

Two new chunk types, SetAudioEntry and NextAudioEntry, were added to allow for multiple audio segments of the same format to be compressed as if they were a single segment (in one compressed audio chunk). This was done in the interest of minimizing protocol overhead.

The **SetAudioEntry** chunk can be used before a compressed audio chunk in **FileData** to indicate that the corresponding relocation entry is at a given offset from the next audio entry. Normally the next audio related relocation entry would be used to locate the position where the data belongs in the file, but if for a example a SetAudioEntry is found and has a value of 2, then 2 audio relocation entries would be skipped when selecting the matching relocation entry. This value is signed, so that negative offsets can be specified.

The **NextAudioEntry** chunk can be used within audio relocation entries in the relocation table. It indicates an offset to the next audio relocation entry in the chain of grouped audio segments. The offset value is unsigned and in reference to the next audio entry.

Note that both offset values are for audio entries only and binary entries are not counted (BinaryChunk and VerbatimChunk).

8.7 Relocation table

The relocation table defines all the segments of audio and binary information so that they can be reconstructed into the original file. The entire RelocTable chunk is compressed using the selected binary compressor (zlib or bzlib). In other words, the RelocTable chunk itself is a block of binary data, the contents of which are the compressed EBML chunks of the relocation table.

The relocation table consists of one or more sub chunks from the list: BinaryChunk, FlacChunk, FlacSplitChunk, WavPackChunk, WavPackSplitChunk and NextAudioEntry which are described below.

8.7.1 BinaryChunk

Indicates a binary segment in the original file and contains only a RawSize field which stores the size of the uncompressed binary data.

8.7.2 FlacChunk and WavPackChunk

Indicates an audio segment in the original file. It contains a RawSize field which stores the original size in bytes of the audio and an optional Flags field which can be used to indicate a different little/big endian or signed/unsigned options. The Flags field is only specified here if the endian and/or sign of the audio format changes within the same file, which is unlikely. These flags can also be set at the FileChunk level, defining the defaults for the entire file or CRAMInfo chunk level for the entire CRAM archive.

8.7.3 FlacSplitChunk and WavPackSplitChunk

Indicates an audio segment in the original file which is split by channel and/or bit width. Channel split audio is when the individual channels are stored separately in the original file, such as 2 mono audio segments being combined to make stereo. Bit width split audio is when the samples themselves are stored as separate blocks. For example: SoundFont 24 bit support stores the 16 most significant bits and the least 8 significant bit portions of each individual sample value in separate areas of the file (thus keeping backwards compatibility with 16 bit SoundFont readers).

These chunks contain the RawSize and Flags field and 3 other fields (all optional): ChanCount, SampleSplit and Offsets.

The **RawSize** field specifies the size in bytes of the first split chunk. For example, if its split stereo then its the size of one channel in bytes. If its sample width split audio (and possibly also channel split) then its the number of bytes of the first split segment.

The **ChanCount** field indicates the channel count for the audio and defaults to the value 2 (Stereo) which is convenient for channel split stereo audio. The value 1 may be specified for bit width split mono audio.

The **SampleSplit** chunk indicates that the individual sample values themselves are split into separate blocks. This chunk contains an integer divided into 3 bit fields (total of 5 bits used) as indicated by the following diagram:

4	3	2	1	0
After	Most Significant		Least Significant	

The **Most Significant (MS)** and **Least Significant (LS)** fields define the **byte** width of the individual sample components. In the SoundFont 24 bit example above, this would be the value 2 for MS (16 bit) and 1 for LS (8 bit). The **After** bit defines if the **MS** audio chunk comes before (**After**=0) or after (**After**=1) the **LS** audio chunk. Currently only a value of 1 or 2 is valid in the **MS** and **LS** fields where the sum is 2 or 3 (for 16 bit or 24 bit split audio respectively).

The **Offsets** chunk provides the offsets for the chunks of split audio data. In the case of channel only split audio (no SampleSplit chunk) the offsets are relative to the end of the previous chunk and Offsets[] contains ChanCount - 1 values (the first chunk is located at offset 0). In the case of audio which is split by bit width (and possibly also by channel) the offsets are stored relative to the end of the first chunk (to provide the needed flexibility in chunk positioning) and Offsets[] contains ChanCount * 2 - 1 values (Chan0:Split1, Chan1:Split0, Chan1:Split1, etc).

Note: If only 1 offset value is expected it can be omitted if its value is 0.

The **SmoothLoop** chunk is used only for lossy WavPack audio chunks and is optional. It defines a loop which can be used during decode to attempt to correct loop artifacts created by the lossy compression (by cross fading or other methods). Currently the reference libInstPatch decoder does not yet use this, although it stores it.

8.8 CRAM hybrid lossless

This feature uses the hybrid encoding mode of WavPack to encode a much smaller lossy CRAM file with a separate correction file which when combined results in the original lossless data. The combined sizes of the files are only slightly larger than the equivalent lossless CRAM file. This provides a nice way for users to preview a potentially large instrument file and then download the rest without incurring much extra overhead.

The only difference between a **.cram** lossless and **.craml** lossy file (besides the obvious audio differences) is the DocType field in the EBML chunk, which is either “CRAM” or “CRAML” respectively and the optional presence of SmoothLoop chunks in the relocation table for the .**craml** case.

8.8.1 CRAM correction file

The **.cramc** correction file on the other hand is lacking many chunks of a regular CRAM file and the DocType is set to “CRAMC”. The other fields of the EBML chunk behave the same as a regular CRAM file except for the Flags field, which currently shouldn't be written (although decoders should ignore it if it exists).

There should be the same number of FileChunks present as in the matching **.craml** file. Each FileChunk should contain a FileData sub chunk with the same count and type of WavPackChunks and/or WavPackSplitChunks containing the WavPack correction data (note that there might not be any sub chunks if the file contains no audio). The last FileChunk should be followed by a FileEnd chunk, as usual.

The FileInfo chunk should be present and contain the FileName and FileSize chunks and are used as an extra check when matching a **.craml** to a **.cramc** file (although the decoder may optionally ignore the situation and attempt to proceed anyways if the FileName and/or FileSize fields don't match between them). These fields can also be used to display what files a correction file is intended for.

All other fields should **not** be present in the correction file, including: RelocTable, RelocTableChksum, FileMD5 and BinaryMD5 as these are taken from the associated **.craml** file.

9 Implementation notes

9.1 EBML chunk handling

If an unrecognized chunk ID is found while decoding a CRAM file, it should be ignored. This allows for backwards compatible additions which don't affect the decoded output. Chunks should be written in the order defined.

9.2 Binary compression

bzlib or zlib are used for compressing binary for CRAM compressed files. The same compressor instance is used for all binary data within a single file (FileData chunk) in a CRAM archive. Separate compressor instances are used for the relocation table and other files in the archive (the compressor context is flushed between uses). This allows for the binary data in separate files and relocation table(s) to be accessed independently and yet maximize compression for all the binary data for a single file. The relocation table entries are in order of position of the original uncompressed file, but the compressed data chunks (in FileData chunk) are not necessarily in the same order. All binary chunks and all audio chunks will be separately in order of position, but binary chunks are only written out when needed (compression buffer is full).

9.3 Audio compression

CRAM utilizes WavPack (default) or FLAC for audio compression. Multiple audio segments can be encoded in the same FLAC or WavPack block (as if they were several audio segments concatenated together). This allows the minimization of encoder overhead and could in some cases be more efficient than encoding many small individual segments separately. In addition, the NextEntryOfs relocation chunk can now be used to indicate that the next audio segment in a chain is not the next audio relocation entry, allowing for all sample data of a particular format in a file to be encoded in the same compressed audio chunk.

9.4 FLAC compression

The compressed FLAC data chunks are written without a FLAC header or STREAMINFO chunk. Only the raw FLAC encoded frames are written. Currently each individual audio segment must use a fixed FLAC block size. This block size may be changed between segments though. Variable block size encoding may be supported in future CRAM revisions, but probably not. Note also that parameters in the FLAC frame header which reference a STREAMINFO chunk may not be used.

9.5 CRAM lossy files

The file MD5 of the original data is also stored in lossy **.craml** files, but the decoder will ignore it if just decoding the lossy file stand alone (although the binary MD5 can, and should, be used in that case to verify the binary portions of the file). It will be used if the **.cramc** is present to verify the resulting original lossless data.